CS 170 First Midterm 20 Feb 2007

NAME (1 pt): ____________________

TA (1 pt): ____________________

Name of Neighbor to your left (1 pt): ____________________

Name of Neighbor to your right (1 pt): ____________________

Instructions: This is a closed book, closed calculator, closed computer, closed network, open brain exam, but you are permited a 1 page, double-sided set of notes, large enough to read without a magnifying glass.

You get one point each for filling in the 4 lines at the top of this page. Each other question is worth 20 points.

Write all your answers on this exam. If you need scratch paper, ask for it, write your name on each sheet, and attach it when you turn it in (we have a stapler).

1	
2	
3	
4	
Total	

2

(1) **(20 points)** The book derived the FFT for vectors of length $n = 2^s$. In this question we will derive the FFT for $n = 3^s$.

- Let $p(z) = \sum_{j=0}^{n-1} p_j \cdot z^j$ be a polynomial of degree at most $n-1$, where $n = 3^s$. Show that $p(z)$ can be written as the sum

$$p(z) = p0(z^3) + z \cdot p1(z^3) + z^2 \cdot p2(z^3) \tag{1}$$

  where $p0(z')$, $p1(z')$ and $p2(z')$ are each polynomials of degree at most $(n/3) - 1$. Be sure to explicitly exhibit the coefficients of each polynomial.

  Answer (3 points):

$$\begin{aligned}
p0(z') &= p_0 + p_3 \cdot z' + p_6 \cdot z'^2 + \cdots + p_{n-3} \cdot z'^{n/3-1} = \sum_{j=0}^{n/3-1} p_{3j} \cdot z'^j \\
p1(z') &= p_1 + p_4 \cdot z' + p_7 \cdot z'^2 + \cdots + p_{n-2} \cdot z'^{n/3-1} = \sum_{j=0}^{n/3-1} p_{3j+1} \cdot z'^j \\
p2(z') &= p_2 + p_5 \cdot z' + p_8 \cdot z'^2 + \cdots + p_{n-1} \cdot z'^{n/3-1} = \sum_{j=0}^{n/3-1} p_{3j+2} \cdot z'^j
\end{aligned}$$

- Let $\omega = e^{2\pi i/n}$, $i = \sqrt{-1}$, be a primitive n-th root of unity. Using equation (1), show that you can evaluate $p(z)$ at the n points ω^0, ω^1, ω^2, ... ,ω^{n-1}, *given* the values of the 3 polynomials $p0(z')$, $p1(z')$ and $p2(z')$ at the $n/3$ points ω^0, ω^3, ω^6, ω^9, ... , ω^{n-3}. You should write down a loop that evaluates $p'_j = p(\omega^j)$, for $j = 0$ to $n-1$, in terms of the values of $p0(z')$, $p1(z')$ and $p2(z')$.

  Answer (5 points):

```
for $j = 0$ to $n/3 - 1$
    $p'_j = p0(\omega^{3j}) + \omega^j \cdot p1(\omega^{3j}) + \omega^{2j} \cdot p2(\omega^{3j})$
    $p'_{j+(n/3)} = p0(\omega^{3j}) + \omega^{j+(n/3)} \cdot p1(\omega^{3j}) + \omega^{2j+2(n/3)} \cdot p2(\omega^{3j})$
    $p'_{j+2(n/3)} = p0(\omega^{3j}) + \omega^{j+2(n/3)} \cdot p1(\omega^{3j}) + \omega^{2j+4(n/3)} \cdot p2(\omega^{3j})$
endfor
```

- Write a recursive subroutine for evaluating $p(z)$ at ω^j, $j = 0, ..., n-1$. Use your answer from the previous part in your answer.

  Answer (7 points):

```
function FFTnp3(p) ... FFT for n a power of 3
    n = length(p)
    if $n = 1$ return p
    p0 =FFTnp3($(p_0, p_3, p_6, ..., p_{n-3})$)
    p1 =FFTnp3($(p_1, p_4, p_7, ..., p_{n-2})$)
    p2 =FFTnp3($(p_2, p_5, p_8, ..., p_{n-1})$)
    $\omega = e^{2\pi\sqrt{-1}/n}$
    ... insert loop from previous part
```

- What is the complexity of your recursive subroutine? You should write down a recurrence for the complexity $T(n)$, justify it, and quote a theorem from class to solve it.

  Answers (5 points): $T(n) = 3T(n/3) + \Theta(n)$ *because each of the 3 recursive calls to FFTnp3 costs* $T(n/3)$*, and the loop over* j *costs* $\Theta(n)$*. By the general theorem about solving recurrences in class (with* $a = b = 3$*,* $c = 1$*), we get that* $T(n) = \Theta(n \log n)$*.*

(1) **(20 points)** You will derive the FFT for vectors of length $m = 3^k$, which will be a variation on the FFT for $m = 2^k$ done in class.

- Let $q(x) = \sum_{j=0}^{m-1} q_j \cdot x^j$ be a polynomial of degree at most $m-1$, where $m = 3^k$. Show that $q(x)$ can be written as the sum

$$q(x) = r(x^3) + x \cdot s(x^3) + x^2 \cdot t(x^3) \tag{2}$$

  where $r(x')$, $s(x')$ and $t(x')$ are each polynomials of degree at most $(m/3) - 1$. Be sure to explicitly exhibit the coefficients of each polynomial.

  Answer (3 points):

$$\begin{aligned} r(x') &= q_0 + q_3 \cdot x' + q_6 \cdot x'^2 + \cdots + q_{m-3} \cdot x'^{m/3-1} = \sum_{j=0}^{m/3-1} q_{3j} \cdot x'^j \\ s(x') &= q_1 + q_4 \cdot x' + q_7 \cdot x'^2 + \cdots + q_{m-2} \cdot x'^{m/3-1} = \sum_{j=0}^{m/3-1} q_{3j+1} \cdot x'^j \\ t(x') &= q_2 + q_5 \cdot x' + q_8 \cdot x'^2 + \cdots + q_{m-1} \cdot x'^{m/3-1} = \sum_{j=0}^{m/3-1} q_{3j+2} \cdot x'^j \end{aligned}$$

- Let $\omega = e^{2\pi i/m}$, $i = \sqrt{-1}$, be a primitive m-th root of unity.
  Using equation (2), show that you can evaluate $q(x)$ at the m points ω^0, ω^1, ω^2, ... ,ω^{m-1}, *given* the values of the 3 polynomials $r(x')$, $s(x')$ and $t(x')$ at the $m/3$ points ω^0, ω^3, ω^6, ω^9, ... , ω^{m-3}.
  Please write down a loop that evaluates $q'_j = q(\omega^j)$, for $j = 0$ to $m-1$, in terms of the values of $r(x')$, $s(x')$ and $t(x')$.

  Answer (5 points):

```
for $j = 0$ to $m/3 - 1$
    $q'_j = r(\omega^{3j}) + \omega^j \cdot s(\omega^{3j}) + \omega^{2j} \cdot t(\omega^{3j})$
    $q'_{j+(m/3)} = r(\omega^{3j}) + \omega^{j+(m/3)} \cdot s(\omega^{3j}) + \omega^{2j+2(m/3)} \cdot t(\omega^{3j})$
    $q'_{j+2(m/3)} = r(\omega^{3j}) + \omega^{j+2(m/3)} \cdot s(\omega^{3j}) + \omega^{2j+4(m/3)} \cdot t(\omega^{3j})$
endfor
```

- Write a recursive procedure to evaluate $q(x)$ at ω^j, $j = 0, ..., m-1$. Use your answer from the previous part in your answer.

  Answer (7 points):

```
function FFTnp3(q) ... FFT for m a power of 3
    $m$ = length(q)
    if $m = 1$ return q
    r =FFTmp3($(q_0, q_3, q_6, ..., q_{m-3})$)
    s =FFTmp3($(q_1, q_4, q_7, ..., q_{m-2})$)
    t =FFTmp3($(q_2, q_5, q_8, ..., q_{m-1})$)
    $\omega = e^{2\pi\sqrt{-1}/m}$
    ... insert loop from previous part
```

- What is the complexity of your recursive subroutine? You should write down a recurrence for the complexity $T(m)$, justify it, and quote a theorem from class to solve it.

  Answers (5 points): $T(m) = 3T(m/3) + \Theta(m)$ *because each of the 3 recursive calls to FFTmp3 costs* $T(m/3)$, *and the loop over* j *costs* $\Theta(m)$. *By the general theorem about solving recurrences in class (with* $a = b = 3$, $c = 1$*), we get that* $T(m) = \Theta(m \log m)$.

(2) **(20 points)** You will derive a divide-and-conquer algorithm for rapid conversion of a number b in binary format to base-9 format. Assume that you have available subroutines for multiplying and dividing (with remainder) n-bit binary numbers that run in $O(n^c)$ bit operations, for some $1 < c \leq 2$.

1. (15 points) Create a divide and conquer algorithm that takes an n-bit binary integer b as input and computes an $O(n)$ base-9 digit representation. Hint: Obtain the top and bottom half of the answer with separate recursions.

   Answer: The base of the recursion will use a subroutine print(b) that takes a binary number in the range $0 \leq b \leq 8$ and prints the corresponding base-9 digit by table look up; it runs in $O(1)$ time.

   The number of base-9 digits in b is $d = \lfloor \log_9 b \rfloor + 1$. This means $b < 9^d$. Let $k = \lceil \frac{d}{2} \rceil$ and divide b by 9^k, getting $b = 9^k \cdot q + r$, with $0 \leq r < 9^k$. Then the d-digit base-9 expansion of b will consist of the $d-k$-digit base-9 expansion of q followed by the k-digit base-9 expansion of r (prepended with enough zeros to make it k digits long). This leads to the recursion:

```
proc Base9Convert(b)
    $d = \lfloor \log_9 b \rfloor + 1$ ... number of base-9 digits in b
        ... ok to overestimate slightly, even by $n = \#bits(b)$,
    call Convert(b, d)
    "erase" any leading zeros in result

proc Convert(b, d)
    if $d \leq 1$, print(b), return, end
    $k = \lceil \frac{d}{2} \rceil$
    $t = 9^k$ ... by repeated squaring, or table look up
    Divide b by t to get $b = t \cdot q + r$, $0 \leq r < 9^k$.
    Convert(q, d - k) ... print leading d - k digits
    Convert(r, k) ... print trailing k digits
```

2. (5 points) Show that it runs in $O(n^c)$ bit operations.

   Answer: $d = \Theta(n)$. Assuming d is a power of 2, we get the recurrence describing the run time for Convert is $T(d) = 2T(d/2) + O(d^c) = O(d^c)$ by the Master Theorem, and $d^c = \Theta(n^c)$. The rest of the algorithm costs at most $O(n)$.

(2) **(20 points)** You will derive a divide-and-conquer algorithm for rapid conversion of a number x in binary format to base-seven format. Assume that you have available subroutines for multiplying and dividing (with remainder) n-bit binary numbers that run in $O(n^c)$ bit operations, for some $1 < c \leq 2$.

1. (15 points) Create a divide and conquer algorithm that takes an n-bit binary integer x as input and computes an $O(n)$ base-7 digit representation. Hint: Obtain the top and bottom half of the answer with separate recursions.

   Answer: The base of the recursion will use a subroutine print(x) that takes a binary number in the range $0 \leq x \leq 6$ and prints the corresponding base-7 digit by table look up; it runs in $O(1)$ time.

   The number of base-7 digits in x is $d = \lfloor \log_7 x \rfloor + 1$. This means $x < 7^d$. Let $k = \lceil \frac{d}{2} \rceil$ and divide x by 7^k, getting $x = 7^k \cdot q + r$, with $0 \leq r < 7^k$. Then the d-digit base-7 expansion of b will consist of the $d - k$-digit base-7 expansion of q followed by the k-digit base-7 expansion of r (prepended with enough zeros to make it k digits long). This leads to the recursion:

```
proc Base7Convert(x)
    $d = \lfloor \log_7 x \rfloor + 1$ ... number of base-7 digits in $x$
        ... ok to overestimate slightly, even by $n = \#bits(x)$,
    call Convert($x, d$)
    "erase" any leading zeros in result

proc Convert($x, d$)
    if $d \leq 1$, print(x), return, end
    $k = \lceil \frac{d}{2} \rceil$
    $t = 7^k$ ... by repeated squaring, or table look up
    Divide $x$ by $t$ to get $x = t \cdot q + r$, $0 \leq r < 7^k$.
    Convert($q, d - k$) ... print leading $d - k$ digits
    Convert($r, k$) ... print trailing $k$ digits
```

2. (5 points) Show that it runs in $O(n^c)$ bit operations.

   Answer: $d = \Theta(n)$. Assuming d is a power of 2, we get the recurrence describing the run time for Convert is $T(d) = 2T(d/2) + O(d^c) = O(d^c)$ by the Master Theorem, and $d^c = \Theta(n^c)$. The rest of the algorithm costs at most $O(n)$.

(3) (**20 points**)

- The first step in RSA is to pick two large primes. This is done by choosing a large random number and testing if it is prime using a primality test. One idea used in primality testing is to use Fermat's Little Theorem to show a number is composite (without actually having the factors!). Using Fermat's Little Theorem, prove that 187 is composite (Hint: you may use the fact that $2^{185} \equiv 87 \bmod 187$).

  Answer: (6 points) 187 is composite since $187 = 11 \times 17$ *but using Fermat's Little theorem you don't need to find this factorization. It suffices to remark that* $2^{186} \equiv 174 \neq 1$ *mod* 187*.*

- Suppose you randomly choose the number 1729, and for every a you try, $a^{1728} \equiv 1 \bmod 1729$. In fact, this is true of every a. What can you conclude about 1729?

  Answer: (4 points) It is either a prime number or a Carmichael number, which is not prime but still satisfies the condition of Fermat's Little Theorem. This one turns out to be a Carmichael number.

- Given the public key $(e, n) = (3, 121)$, encrypt the message $m = 5$.

  Answer: (5 points) $m^e \equiv 4$ *mod* 121*.*

- We will see later in the course that quantum computers, if we managed to build them, would be able to factor large numbers rapidly (in polynomial time). Why and how would this allow you to "break" RSA? More precisely, say a message m has been encoded using the public key (e, n), how do you efficiently recover m from the encrypted message c?

  Answer: (5 points) You can use the quantum computer to factor n into its factors p and q, from which you can compute the decryption key d by computing the inverse of e modulo $(p-1)(q-1)$*. This inversion is based on the extended version of Euclid's algorithm, which runs fast (in cubic time). From the decryption key we recover the message* $m \equiv c^d$ *mod* n*. This exponentiation modulo* n *is also fast.*

(3) (**20 points**)

- The first step in RSA is to pick two large primes. This is done by choosing a large random number and testing if it is prime using a primality test. One idea used in primality testing is to use Fermat's Little theorem to show a number is composite (without actually having the factors!). Using Fermat's Little Theorem, prove that 221 is composite (Hint: you may use the fact that $2^{218} \equiv 4 \bmod 221$).

  Answer: (6 points) 221 is composite since $221 = 13 \times 17$ *but using Fermat's Little theorem you don't need to find this factorization. It suffices to remark that* $2^{220} \equiv 16 \neq 1$ *mod* 221.

- Suppose you randomly choose the number 1105, and for every a you try, $a^{1104} \equiv 1 \bmod 1105$. In fact, this is true of every a. What can you conclude about 1105?

  Answer: (4 points) It is either a prime number or a Carmichael number, which is not prime but still satisfies the condition of Fermat's Little Theorem. This one turns out to be a Carmichael number.

- Given the public key $(e, n) = (3, 85)$, encrypt the message $m = 10$.

  Answer: (5 points) $m^e \equiv 65$ *mod* 85.

- We will see later in the course that quantum computers, if we managed to build them, would be able to factor large numbers rapidly (in polynomial time). Why and how would this allow you to "break" RSA? More precisely, say a message m has been encoded using the public key (e, n), how do you efficiently recover m from the encrypted message c?

  Answer: (5 points) You can use the quantum computer to factor n into its factors p and q, from which you can compute the decryption key d by computing the inverse of e modulo $(p-1)(q-1)$. *This inversion is based on the extended version of Euclid's algorithm, which runs fast (in cubic time). From the decryption key we recover the message* $m \equiv c^d$ *mod n. This exponentiation modulo n is also fast.*

(4) **True/False Questions. (20 points)** Circle the correct (best) answer. No explanation is required. Each correct answer is worth +2 points, Incorrect answers are worth −2 points (i.e. 2 points will be **deducted**), and omitted questions are worth 0 points (no points given or deducted). Therefore, only answer if you are reasonably certain. For all algorithms described below, assume their instructions execute sequentially on a single processor.

T or F: $\sum_{i=1}^{n} i^r (\log i)^s = \Theta(n^{r+1}(\log n)^s)$.

Answer: **True:** *$\sum_{i=1}^{n} i^r(\log i)^s \le \sum_{i=1}^{n} n^r(\log n)^s = n^{r+1}(\log n)^s$, and $\sum_{i=1}^{n} i^r(\log i)^s \ge \sum_{i=n/2}^{n} i^r(\log i)^s \ge \sum_{i=n/2}^{n}(\frac{n}{2})^r(\log\frac{n}{2})^s = (\frac{n}{2})^{r+1}(\log\frac{n}{2})^s = (\frac{1}{2})^{r+1}n^{r+1}(\log n - \log 2)^s \ge (\frac{1}{2})^{r+1}n^{r+1}(\frac{1}{2}\log n)^s$ (for $n \ge 4$) $= \alpha n^{r+1}(\log n)^s \Rightarrow \sum_{i=1}^{n} i^r(\log i)^s = \Theta(n^{r+1}(\log n)^s)$.*

T or F: If $T(n) = 9T(n-2) + 3$, with $T(2) = T(1) = \Theta(1)$, then $T(n) = \Theta(3^n \log n)$.

Answer: **False:** *$T(n) = 3(1 + 9 + 9^2 + \ldots + 9^k) = 3 \cdot \frac{9^{k+1}-1}{9-1}$, where k is the number of recursive calls until we reach either $n = 2$ or $n = 1$. Since $k = \frac{n}{2}$, $T(n) = \frac{3}{8}(9^{\frac{n}{2}+1} - 1) = \frac{3}{8}(9 \cdot 3^n - 1) = \Theta(3^n)$.*

T or F: The worst-case running time of quicksort is asymptotically slower than the worst-case running time of mergesort, but quicksort is preferred in practice because of its fast average case and small constant hidden in the asymptotic runtime.

Answer: **True:** *In the worst-case, mergesort runs in $\Theta(n \log n)$ time and quicksort runs in $\Theta(n^2)$ time. On average, however, quicksort runs in $\Theta(n \log n)$ with a relatively small hidden constant.*

T or F: $n^{2+\frac{1}{\sqrt{n}}} = O(n^2)$.

Answer: **True:** *$\frac{n^{2+\frac{1}{\sqrt{n}}}}{n^2} \to 1$ as $n \to \infty$, since $\log(n^{\frac{1}{\sqrt{n}}}) = \frac{\log n}{\sqrt{n}} \to 0$, implying $n^{2+\frac{1}{\sqrt{n}}} = \Theta(n^2) \Rightarrow n^{2+\frac{1}{\sqrt{n}}} = O(n^2)$.*

T or F: If $F(n)$ is the running time of the Fast Fourier Transform on an n-element vector, and $G(n)$ is the running time of Strassen's algorithm when multiplying two $n \times n$ general matrices, then $(F(n))^2$ is $O(G(n))$.

Answer: **True:** *$F(n) = \Theta(n \log n)$, and $G(n) = \Theta(n^{\log_2 7}) \Rightarrow (F(n))^2 = \Theta(n^2 \log^2 n) = O(n^{\log_2 7}) = O(G(n))$.*

T or F: If $T(n) = 16T(n/4) + n^2$, with $T(1) = \Theta(1)$, then $T(n) = \Theta(n^2)$.

Answer: **False:** *$n^{\log_b a} = n^2 = f(n)$, so the Master Theorem gives $T(n) = \Theta(n^2 \log n)$.*

T or F: $n! = \Theta(n^n)$.

Answer: **False:** $\frac{n^n}{n!} = n \cdot \frac{n^{(n-1)}}{n!} = n \cdot (\frac{n}{n} \cdot \frac{n}{n-1} \cdot \ldots \cdot \frac{n}{2}) > n$ *when* $n > 2 \Rightarrow \frac{n^n}{n!} \to \infty$ *as* $n \to \infty \Rightarrow n!$ *is* $O(n^n)$ *but not* $\Omega(n^n) \Rightarrow n!$ *is not* $\Theta(n^n)$.

T or F: $\sum_{i=1}^{n}(6i^2 - 2i) = 2n^2(n+1)$.

Answer: **True:** $\sum_{i=1}^{n}(6i^2 - 2i) = 6\frac{(n)(n+1)(2n+1)}{6} - 2\frac{n(n+1)}{2} = 2n^2(n+1)$, *or by induction.*

T or F: $3^{(2^n)} = \Omega(2^{(3^n)})$.

Answer: **False:** *Since* $\frac{\log(2^{(3^n)})}{\log(3^{(2^n)})} = \frac{3^n \log 2}{2^n \log 3} = \frac{\log 2}{\log 3}(\frac{3}{2})^n \to \infty$ *as* $n \to \infty$, $\frac{2^{(3^n)}}{3^{(2^n)}} \to \infty$ *as* $n \to \infty \Rightarrow 3^{(2^n)}$ *is* $O(2^{(3^n)})$ *but not* $\Omega(2^{(3^n)})$.

T or F: Suppose you are writing a divide-and-conquer algorithm that does an amount of work costing $\Theta(n\log(n))$ and then splits the problem of size n into six subproblems of size $n/3$. Your algorithm would be asymptotically faster if you reduced the $\Theta(n\log(n))$ cost to $\Theta(\log n)$.

Answer: **False:** *Initial recurrence relation is* $T(n) = 6T(n/3) + \Theta(n\log(n))$, *which has solution* $T(n) = \Theta(n^{\log_3 6})$ *by the Master Theorem. Changing the algorithm gives a new recurrence of* $T(n) = 6T(n/3) + \Theta(\log(n))$, *but it has the same solution of* $T(n) = \Theta(n^{\log_3 6})$ *by the Master Theorem.*

(4) **True/False Questions. (20 points)** Circle the correct (best) answer. No explanation is required. Each correct answer is worth +2 points, Incorrect answers are worth −2 points (i.e. 2 points will be **deducted**), and omitted questions are worth 0 points (no points given or deducted). Therefore, only answer if you are reasonably certain. For all algorithms described below, assume their instructions execute sequentially on a single processor.

T or F: $\sum_{i=1}^{n}(6i^2 - 2i) = 2n^2(n+1)$.

Answer: **True:** $\sum_{i=1}^{n}(6i^2-2i) = 6\frac{(n)(n+1)(2n+1)}{6} - 2\frac{n(n+1)}{2} = 2n^2(n+1)$ *or by induction.*

T or F: If $T(n) = 16T(n/4) + n^2$, with $T(1) = \Theta(1)$, then $T(n) = \Theta(n^2)$.

Answer: **False:** $n^{\log_b a} = n^2 = f(n)$, *so the Master Theorem gives* $T(n) = \Theta(n^2 \log n)$.

T or F: $n^{2+\frac{1}{\sqrt{n}}} = O(n^2)$.

Answer: **True:** $\frac{n^{2+\frac{1}{\sqrt{n}}}}{n^2} \to 1$ *as* $n \to \infty$, *since* $\log(n^{\frac{1}{\sqrt{n}}}) = \frac{\log n}{\sqrt{n}} \to 0$, *implying* $n^{2+\frac{1}{\sqrt{n}}} = \Theta(n^2) \Rightarrow n^{2+\frac{1}{\sqrt{n}}} = O(n^2)$.

T or F: $n! = \Theta(n^n)$.

Answer: **False:** $\frac{n^n}{n!} = n \cdot \frac{n^{(n-1)}}{n!} = n \cdot (\frac{n}{n} \cdot \frac{n}{n-1} \cdot \ldots \cdot \frac{n}{2}) > n$ *when* $n > 2 \Rightarrow \frac{n^n}{n!} \to \infty$ *as* $n \to \infty \Rightarrow n!$ *is* $O(n^n)$ *but not* $\Omega(n^n) \Rightarrow n!$ *is not* $\Theta(n^n)$.

T or F: Suppose you are writing a divide-and-conquer algorithm that does an amount of work costing $\Theta(n\log(n))$ and then splits the problem of size n into six subproblems of size $n/3$. Your algorithm would be asymptotically faster if you reduced the $\Theta(n\log(n))$ cost to $\Theta(\log n)$.

Answer: **False:** *Initial recurrence relation is* $T(n) = 6T(n/3) + \Theta(n\log(n))$, *which has solution* $T(n) = \Theta(n^{\log_3 6})$ *by the Master Theorem. Changing the algorithm gives a new recurrence of* $T(n) = 6T(n/3) + \Theta(\log(n))$, *but it has the same solution of* $T(n) = \Theta(n^{\log_3 6})$ *by the Master Theorem.*

T or F: If $T(n) = 9T(n-2) + 3$, with $T(2) = T(1) = \Theta(1)$, then $T(n) = \Theta(3^n \log n)$.

Answer: **False:** $T(n) = 3(1 + 9 + 9^2 + \ldots + 9^k) = 3 \cdot \frac{9^{k+1}-1}{9-1}$, *where* k *is the number of recursive calls until we reach either* $n = 2$ *or* $n = 1$. *Since* $k = \frac{n}{2}$, $T(n) = \frac{3}{8}(9^{\frac{n}{2}+1} - 1) = \frac{3}{8}(9 \cdot 3^n - 1) = \Theta(3^n)$.

T or F: $3^{(2^n)} = \Omega(2^{(3^n)})$.

Answer: **False:** $T(n) = 3(1 + 9 + 9^2 + \ldots + 9^k) = 3 \cdot \frac{9^{k+1}-1}{9-1}$, *where* k *is the number of recursive calls until we reach either* $n = 2$ *or* $n = 1$. *Since* $k = \frac{n}{2}$, $T(n) = \frac{3}{8}(9^{\frac{n}{2}+1} - 1) = \frac{3}{8}(9 \cdot 3^n - 1) = \Theta(3^n)$.

T or F: The worst-case running time of quicksort is asymptotically slower than the worst-case running time of mergesort, but quicksort is preferred in practice because of its fast average case and small constant hidden in the asymptotic runtime.

Answer: **True:** *In the worst-case, mergesort runs in $\Theta(n \log n)$ time and quicksort runs in $\Theta(n^2)$ time. On average, however, quicksort runs in $\Theta(n \log n)$ with a relatively small hidden constant.*

T or F: $\sum_{i=1}^{n} i^r (\log i)^s = \Theta(n^{r+1} (\log n)^s)$.

Answer: **True:** $\sum_{i=1}^{n} i^r (\log i)^s \le \sum_{i=1}^{n} n^r (\log n)^s = n^{r+1} (\log n)^s$, *and* $\sum_{i=1}^{n} i^r (\log i)^s \ge \sum_{i=n/2}^{n} i^r (\log i)^s \ge \sum_{i=n/2}^{n} (\frac{n}{2})^r (\log \frac{n}{2})^s = (\frac{n}{2})^{r+1} (\log \frac{n}{2})^s = (\frac{1}{2})^{r+1} n^{r+1} (\log n - \log 2)^s \ge (\frac{1}{2})^{r+1} n^{r+1} (\frac{1}{2} \log n)^s$ *(for $n \ge 4$)* $= \alpha n^{r+1} (\log n)^s \Rightarrow \sum_{i=1}^{n} i^r (\log i)^s = \Theta(n^{r+1} (\log n)^s)$.

T or F: If $F(n)$ is the running time of the Fast Fourier Transform on an n-element vector, and $G(n)$ is the running time of Strassen's algorithm when multiplying two $n \times n$ general matrices, then $(F(n))^2$ is $O(G(n))$.

Answer: **True:** $F(n) = \Theta(n \log n)$, *and* $G(n) = \Theta(n^{\log_2 7}) \Rightarrow (F(n))^2 = \Theta(n^2 \log^2 n) = O(n^{\log_2 7}) = O(G(n))$.